MadAnalysis 5 v1.6
Expert-Mode reference card

November 19, 2018

website: http://madanalysis.irmp.ucl.ac.be/

references: arXiv:1206.1599[hep-ph], arXiv:1405.3982[hep-ph], arXiv:1407.3278[hep-ph]

1 Creating an analysis template in the expert mode

For sophisticated analysis going beyond the capabilities of the normal mode of running of MadAnalysis 5, the user has to rely on the so-called expert mode, where analyses are implemented directly in the C++ framework of the platform. A blank template analysis can be generated by typing, in a shell,

```
bin/ma5 --expert      bin/ma5 -e      bin/ma5 -E
```

MadAnalysis 5 then asks information about the name of the working directory to create, as well as information on the name of the analysis class that will have to be designed. Those two pieces of information can be provided as arguments when calling MadAnalysis 5 from the shell,

```
bin/ma5 -E <dirname> <analysis>
```

where `<dirname>` consists in the working directory name, and `analysis` the name of the analysis class.

For a proper use of the program, the user has to setup some environment variables accordingly. This can be done by entering the `Build` subfolder of the working directory and typing in a shell,

```
source setup.sh         source setup.csh
```

depending on the shell nature. The `Build` folder also contains a makefile allowing for standard `make` commands,

```
make clean       make proper       make
```

The first command allows one to remove all intermediate object and backup files whilst the second command yields the removal of all files that have been created by the make action. Finally, the last command allows to build the code.

The code can then be run from the `Build` directory by typing,

```
MadAnalysis5Job [options] [inputfile]
```

The file `[inputfile]` consists in a text file with a list of paths pointing to the event samples to analyze, with one filename per line. Several options are available, as summarized in the table below.

Option	Description
`--check_event`	Sanity check of the input file.
`--no_event_weight`	Ignores the event weights.
`--ma5_version=XXXX`	Allows to specify which version of the MadAnalysis 5 console to use.

2 Portable datatypes

The SampleAnalyzer data format includes several portable data types, that we show in the table below together with the corresponding bit width.

Name	Bit width	Description
`MAbool`	1	Boolean.
`MAint8`	8	Byte integer.
`MAint16`	16	Short integer.
`MAint32`	32	Integer.
`MAint64`	64	Long integer.
`MAuint8`	8	Unsigned byte integer.
`MAuint16`	16	Unsigned short integer.
`MAuint32`	32	Unsigned integer.
`MAuint64`	64	Unsigned long integer.
`MAfloat32`	32	Single-precision floating-point number.
`MAfloat64` or `MAdouble64`	64	Double-precision floating-point number.

Moreover, four-vectors can be implemented as instances of the `MALorentzVector` class that contains the same methods as the Root `TLorentzVector` class [3].

3 Data format for an event sample

The analysis class contains an `Initialize`, an `Execute` and a `Finalize` method that are respectively executed before starting to read an event sample, on each event and after having read all events. The `Execute` method requires two arguments, an instance of the `SampleFormat` class and an event passed as an `EventFormat` instance (see the next subsection). Monte Carlo event samples are generally accompanied with global information on the sample, such as the identifier of the parton density set that has been used or the total cross section associated with the described process. Those pieces of information are stored as attributes of the above-mentioned `SampleFormat` object, and can be retrieved through the methods given in the table below (in particular through attributes of the `mc()` and `rec()` objects) on run time.

Method name and type	Description
`MCSampleFormat* mc()`	Monte Carlo information (see below).
`RecSampleFormat* rec()`	Reconstruction information (see below).
`const std::string name()`	Sample name.
`const MAuint64& nevents() const`	Number of events in the sample.
`const std::vector<std::string>& header() const`	Sample header.

The Monte Carlo information is available through the `mc()` object, a pointer to an instance of the `MCSampleFormat` class whose attributes are given in the table below.

Method name and type	Description
`const std::pair<MAint32,MAint32>& beamPDGID() const`	PDG codes of the intial partons.
`const std::pair<MAfloat64,MAfloat64>& beamE() const`	Beam energy.
`const std::pair<MAuint32,MAuint32>& beamPDFauthor() const`	Group associated with the used parton density set.
`const std::pair<MAuint32,MAuint32>& beamPDFID() const`	Identifier of the used parton density set.
`const MAint32& weightMode() const`	Information on the event weights.
`const MAfloat64& xsection() const`	Cross section associated with the sample.
`const MAfloat64& xsection_error() const`	Uncertainty on the sample cross section.
`const MAfloat64& sumweight_positive() const`	Sum of the weights of all positively-weighted events.
`const MAfloat64& sumweight_negative() const`	Sum of the weights of all negatively-weighted events.
`const std::vector<ProcessFormat>& processes()`	List of all described processes (see below).
`const WeightDefinition& weight_definition()`	Definitions of the different weights for events featuring multiple weights.

Those attributes rely on two classes, the `ProcessFormat` one allowing for the description of a physical process and the `WeightDefinition` one connected to the potential assignment of multiple weights to a given event [1]. Whilst the following methods have been implemented for the former class,

Method name and type	Description
`const MAfloat64& xsection() const`	Associated cross section.
`const MAfloat64& xsectionError() const`	Error on the cross section.
`const MAfloat64& weightMax() const`	Maximum weight for an event.
`const MAuint32& processId() const`	Process identification number.

the latter class only allows for listing the names of the set of weights associated with each event,

```
void Print() const
```

The `rec()` method of the `MCSampleFormat` class consists in an instance of the `RecSampleFormat` class that does not come with any built-in method. This is left for future developments.

4 Data format for an event

An event object possesses two attributes `mc()` and `rec()` that are this time instances of the `MCEventFormat` and `RecEventFormat` classes respectively. These two classes are respectively connected to Monte Carlo events (as simulated by a Monte Carlo event generator) and reconstructed events as obtained after gathering all final-state objects into reconstructed physical objects to be used for specific analyses. The `MCEventFormat` class comes with the methods summarized in the following table.

`const MCParticleFormat& MET() const` `MCParticleFormat& MET() const`	Missing transverse energy
`const MCParticleFormat& MHT() const` `MCParticleFormat& MHT() const`	Missing transverse hadronic energy.
`const MAfloat64& TET() const` `MAfloat64& TET() const`	Total (visible) transverse energy.
`const MAfloat64& THT() const` `const MAfloat64& THT() const`	Total hadronic transverse energy.
`const MAfloat64& Meff() const` `MAfloat64& Meff() const`	Effective mass $\left(\sum_{\text{jets}} p_T + \not{E}_T\right)$.
`const MAfloat64& alphaQED() const`	Used value for the electromagnetic coupling.
`const MAfloat64& alphaQCD() const`	Used value for the strong coupling.
`const WeightCollection& multiweights() const`	Container for all event weights.
`const std::vector<MCParticleFormat>& particles() const`	All particles of the event.
`const MAuint32& processId() const`	Identifier of the physical process related to the event.
`const MAfloat64& scale() const`	Factorization scale choice.
`const MAfloat64& weight() const`	Event weight.

Weights (as returned by the `multiweights()` method) are stored as an instance of the `WeightCollection` class, which comes with the methods given in the table below.

`const MAfloat64& Get(MAuint32 id) const`	Weight value corresponding to the weight identifier `id`.
`const std::map<MAuint32,MAfloat64>& GetWeights() const`	The full list of weights as identifier-value pairs.

As indicated above, the event particle content can be obtained via the method `particles()` of the `MCEventFormat` class. This returns a vector of `MCParticleFormat` objects, each element corresponding to an initial-state, a final-state or an intermediate-state particle. This class inherits all methods available from the `ParticleBaseFormat` class, that are collected in the table below.

`MALorentzVector& momentum()`	
`const MALorentzVector& momentum() const`	
	Four-momentum.
`const MAfloat32 beta() const`	Velocity (in c units).
`const MAfloat32 e() const`	Energy.
`const MAfloat32 et() const`	Transverse energy.
`const MAfloat32 eta() const`	Pseudorapidity.
`const MAfloat32 abseta() const`	Pseudorapidity in absolute value.
`const MAfloat32 gamma() const`	Lorentz factor.
`const MAfloat32 m() const`	Invariant mass.
`const MAfloat32 mt() const`	Transverse mass.
`const MAfloat32 phi() const`	Azimuthal angle.
`const MAfloat32 p() const`	Magnitude of the momentum.
`const MAfloat32 pt() const`	Transverse momentum.
`const MAfloat32 px() const`	x-component of the momentum.
`const MAfloat32 py() const`	y-component of the momentum.
`const MAfloat32 pz() const`	z-component of the momentum.
`const MAfloat32 r() const`	Position in the $\eta - \phi$ plane.
`const MAfloat32 theta() const`	Polar angle.
`const MAfloat32 y() const`	Rapidity.

The `ParticleBaseFormat` class also includes a set of methods that involve the particle itself as well as another object. These are given in the following table.

`const MAfloat32 angle(const ParticleBaseFormat& p) const`	
`const MAfloat32 angle(const ParticleBaseFormat* p) const`	
	Angular separation between the particle momentum and the momentum of another particle `p`.
`const MAfloat32 mt_met(const MALorentzVector& MET) const`	
	Transverse mass of the system made of the particle and the missing momentum.
`const MAfloat32 dphi_0_pi(const ParticleBaseFormat* p) const`	
`const MAfloat32 dphi_0_pi(const ParticleBaseFormat& p) const`	
	Azimuthal separation between the particle momentum and the momentum of another particle `p`, normalized in $[0, \pi]$.
`const MAfloat32 dphi_0_2pi(const ParticleBaseFormat* p) const`	
`const MAfloat32 dphi_0_2pi(const ParticleBaseFormat& p) const`	
	Azimuthal separation between the particle momentum and the momentum of another particle `p`, normalized in $[0, 2\pi]$.
`const MAfloat32 dr(const ParticleBaseFormat& p) const`	
`const MAfloat32 dr(const ParticleBaseFormat* p) const`	
	Angular distance, in the $\eta - \phi$ plane, between the particle momentum and the momentum of another particle `p`.

Moreover, the `MCParticleFormat` class includes the extra methods listed below.

`const MAfloat64& ctau() const`	Particle decay length.
`const MAbool& isPU()  const`	Tests whether the particle belongs to the pile-up event.
`const MAint32& pdgid()  const`	PDG identifier of the particle.
`const MAfloat32& spin() const`	Cosine of the angle between the particle momentum and its spin vector, computed in the laboratory frame.
`const MAint16& statuscode() const`	Code indicating the particle initial-, intermediate- or final-state nature.
`const std::vector<MCParticleFormat*>& daughters() const`	Particles in which the current particle decays into.
`const std::vector<MCParticleFormat*>& mothers() const`	Particles from which the current particle originates from.
`const MALorentzVector& decay_vertex() const`	Spacetime position of the particle decay.

A slightly different format is available for reconstructed events. An event is here provided as an instance of the `RecEventFormat` class, which comes with the following methods.

`const std::vector<RecPhotonFormat>& photons() const`
: Reconstructed photons.

`const std::vector<RecLeptonFormat>& electrons() const`
: Reconstructed electrons.

`const std::vector<RecLeptonFormat>& muons() const`
: Reconstructed muons.

`const std::vector<RecTauFormat>& taus() const`
: Reconstructed hadronic taus.

`const std::vector<RecJetFormat>& fatjets() const`
: Reconstructed fat jets.

`const std::vector<RecJetFormat>& jets() const`
: Reconstructed jets.

`const std::vector<RecJetFormat>& genjets() const`
: Parton-level jets.

`const std::vector<RecTrackFormat>& tracks() const`
: Tracks left in a detector.

`const std::vector<RecTowerFormat>& towers() const`
: Calorimetric deposits left in a detector.

`const std::vector<RecTrackFormat>& EFlowTracks() const`
: Tracks left in a detector, reconstructed from the particle flow information.

`const std::vector<RecParticleFormat>& EFlowPhotons() const`
: Photons, reconstructed from the particle flow information.

`const std::vector<RecParticleFormat>& EFlowNeutralHadrons() const`
: Neutral hadrons, reconstructed from the particle flow information.

`const RecParticleFormat& MET() const`
: Missing transverse energy.

`const RecParticleFormat& MHT() const`
: Missing hadronic energy.

`const MAfloat64& TET() const`
: Visible transverse energy.

`const MAfloat64& THT() const`
: Hadronic transverse energy.

`const MAfloat64& Meff() const`
: Effective mass $\left(\sum_{\text{jets}} p_T + \not{E}_T\right)$.

`const std::vector<const MCParticleFormat*>& MCHadronicTaus() const`
: Parton-level taus that decayed hadronically.

`const std::vector<const MCParticleFormat*>& MCElectronicTaus() const`
: Parton-level taus that decayed into an electron and missing energy.

`const std::vector<const MCParticleFormat*>& MCMuonicTaus() const`
: Parton-level taus that decayed into a muon and missing energy.

`const std::vector<const MCParticleFormat*>& MCBquarks() const`
: Parton-level b-quarks of the event.

`const std::vector<const MCParticleFormat*>& MCCquarks() const`
: Parton-level c-quarks of the event.

The above methods introduce the data format implemented for all reconstructed objects. It relies on various types (`RecLeptonFormat`, `RecJetFormat`, `RecPhotonFormat`, `RecTauFormat`, `RecTrackFormat` and `RecTowerFormat`) that inherit from the `RecParticleFormat` class based on the `BaseParticleFormat` class (see above). All these new classes include the following set of methods.

`const MAfloat32& EEoverHE() const`

Ratio of the electromagnetic to hadronic calorimetric energy associated with the object.

`const MAfloat32& HEoverEE() const`

Ratio of the hadronic to electromagnetic calorimetric energy associated with the object.

`const MAint32 DecayMode() const`

Identifier of the decay mode of a `RecTauFormat` object. The available choices are 1 ($\tau \to e\nu\nu$), 2 ($\tau \to \mu\nu\nu$), 3 ($\tau \to K\nu$), 4 ($\tau \to K^*\nu$), 5 ($\tau \to \rho(\to \pi\pi^0)\nu$), 6 ($\tau \to a_1(\to \pi\pi^0\pi^0)\nu$), 7 ($\tau \to a_1(\to \pi\pi\pi)\nu$), 8 ($\tau \to \pi\nu$), 9 ($\tau \to \pi\pi\pi\pi^0\nu$) and 0 (any other decay mode).

`const MAbool& btag() const`

Indicates whether a `RecJetFormat` object has been b-tagged.

`const MAbool& ctag() const`

Indicates whether a `RecJetFormat` object has been c-tagged.

`const std::vector<MAint32>& constituents() const`

Returns the constituents of a `RecJetFormat` object.

`const int charge() const`

The electric charge of the object. This method is available for the `RecLeptonFormat`, `RecTauFormat` and `RecTrackFormat` classes.

`MAfloat32 d0() const`

Impact parameter of a `RecLeptonFormat` object. An extension to the other classes of reconstructed objects is foreseen.

`MAfloat32 d0error() const`

Uncertainty on the impact parameter of a `RecLeptonFormat` object.

`const MAuint16 ntracks() const`

Number of charged tracks associated with a reconstructed object. This method is available for the `RecJetFormat` and `RecTauFormat` classes.

`MAbool isElectron() const`

Indicates if a `RecLeptonFormat` object is an electron.

`MAbool isMuon() const`

Indicates if a `RecLeptonFormat` object is a muon.

`const MAbool& true_btag() const`

Indicates whether a `RecJetFormat` object is a true b-jet.

`const MAbool& true_ctag() const`

Indicates whether a `RecJetFormat` object is a true c-jet.

---

5 Lepton and photon isolation

In the context of reconstructed events, lepton and photon isolation can be ensured by relying either on track information, on calorimetric information, on the combination of both or on a reconstruction based on the energy flow. Corresponding isolation methods are available, within the MadAnalysis 5 data format, through the four classes,

```
PHYSICS->Isol->tracker          PHYSICS->Isol->calorimeter
PHYSICS->Isol->combined         PHYSICS->Isol->eflow
```

respectively connected to the four ways to enforce object isolation. All those classes come with two methods summarized in the table below.

```
MAfloat64 relIsolation(const <x>& prt,const RecEventFormat* evt,
       const double& DR, double PTmin=0.5) const
```

Sum of the transverse momenta of all objects lying in a cone of radius `DR` centered on the considered object `prt`, and whose transverse momentum is larger than `PTmin`. The sum is evaluated relatively to the transverse momentum of the considered object `prt` that can be either a `RecLeptonFormat` or a `RecPhotonFormat` object (*i.e.* the value of the `<x>` type).

```
MAfloat64 sumIsolation(const <x>& prt,const RecEventFormat* evt,
       const double& DR, double PTmin=0.5) const
```

Sum of the transverse momenta of all objects lying in a cone of radius `DR` centered on the considered object `prt`, and whose transverse momentum is larger than `PTmin`. The object `prt` can be either a `RecLeptonFormat` or a `RecPhotonFormat` object (*i.e.* the value of the `<x>` type).

When isolation is imposed on the basis of the energy flow (`PHYSICS->Isol->eflow`), those methods take an extra argument,

```
MAfloat64 relIsolation(const <x>& prt, const RecEventFormat* evt,
    const double& DR, double PTmin=0.5, ComponentType type) const

MAfloat64 sumIsolation(const <x>& prt, const RecEventFormat* evt,
    const double& DR, double PTmin=0.5, ComponentType type) const
```

where `type` can take one of the four values,

```
TRACK_COMPONENT                 PHOTON_COMPONENT
NEUTRAL_COMPONENT               ALL_COMPONENTS
```

In the first case, the activity around the considered object `prt` is evaluated only from the charged track information, whilst in the second case, only the photon information is considered. In the third case, the neutral hadron activity is accounted for whilst the last option consists in the sum of the three previous cases.

In addition, a series of `JetCleaning` functions are provided in the aim of cleaning jet collections from objects present in a lepton collection or a photon collection.

```
std::vector<const RecJetFormat*> PHYSICS->Isol->JetCleaning(
   const std::vector<const RecJetFormat*>& uncleaned,
   const std::vector<const RecLeptonFormat*>& leptons,
   double DeltaRmax = 0.1, double PTmin = 0.5) const
std::vector<const RecJetFormat*> PHYSICS->Isol->JetCleaning(
   const std::vector<RecJetFormat>& uncleaned,
   const std::vector<const RecLeptonFormat*>& leptons,
   double DeltaRmax = 0.1, double PTmin = 0.5) const
```

Removal from the `uncleaned` jet collection of all leptons included in the `leptons` collection lying at an angular distance of at most `DeltaRmax` of the jet, and whose transverse momentum is of at least `PTmin`.

```
std::vector<const RecJetFormat*> PHYSICS->Isol->JetCleaning(
   const std::vector<const RecJetFormat*>& uncleaned,
   const std::vector<const RecPhotonFormat*>& photons,
   double DeltaRmax = 0.1, double PTmin = 0.5) const
std::vector<const RecJetFormat*> PHYSICS->Isol->JetCleaning(
   const std::vector<RecJetFormat>& uncleaned,
   const std::vector<const RecPhotonFormat*>& photons,
   double DeltaRmax = 0.1, double PTmin = 0.5) const
```

Removal from the `uncleaned` jet collection of all photons included in the `photons` collection lying at an angular distance of at most `DeltaRmax` of the jet, and whose transverse momentum is of at least `PTmin`.

6 Observables

The SampleAnalyzer data format contains various methods to compute observables connected to the entire event, which includes in particular a small set of common transverse variables and a set of methods related to object identification. They are available through a series of `PHYSICS` services, that first contain the two general methods below.

```
MAint32 PHYSICS->GetTauDecayMode (const MCParticleFormat* part)
```

Returns the identifier of the decay mode of a tau particle. The available values are 1 ($e\nu\nu$), 2 ($\mu\nu\nu$), 3 ($K\nu$), 4 ($K^*\nu$), 5 ($\rho(\to \pi\pi^0)\nu$), 6 ($a_1(\to \pi\pi^0\pi^0)\nu$), 7 ($a_1(\to \pi\pi\pi)\nu$), 8 ($\pi\nu$), 9 ($\pi\pi\pi\pi^0\nu$) and 0 (any other decay mode).

```
double PHYSICS->SqrtS(const MCEventFormat* event) const
```

Returns the partonic center-of-mass energy.

Identification functions are collected as methods attached to the `PHYSICS->Id` object. The list of available methods is given in the table below.

`MAbool IsInitialState(const MCParticleFormat& part) const`
`MAbool IsInitialState(const MCParticleFormat* part) const`
Tests the initial-state nature of an object.
`MAbool IsInterState(const MCParticleFormat* part) const`
`MAbool IsInterState(const MCParticleFormat& part) const`
Tests the intermediate-state nature of an object.
`MAbool IsFinalState(const MCParticleFormat& part) const`
`MAbool IsFinalState(const MCParticleFormat* part) const`
Tests the final-state nature of an object.
`bool IsHadronic(const RecParticleFormat* part) const`
`bool IsHadronic(const MCParticleFormat* part) const`
`bool IsHadronic(MAint32 pdgid) const`
Tests the hadronic nature of an object.
`bool IsInvisible(const RecParticleFormat* part) const`
`bool IsInvisible(const MCParticleFormat* part) const`
Tests the invisible nature of an object.
`MAbool IsBHadron(MAint32 pdg)`
`MAbool IsBHadron(const MCParticleFormat& part)`
`MAbool IsBHadron(const MCParticleFormat* part)`
Tests whether the object is a B-hadron.
`MAbool IsCHadron(MAint32 pdg)`
`MAbool IsCHadron(const MCParticleFormat& part)`
`MAbool IsCHadron(const MCParticleFormat* part)`
Tests whether the object is a C-hadron.

Finally, a set of transverse variables can be evaluated by relying on the `PHYSICS->Transverse` object. The following methods are available within the SampleAnalyzer data format.

`double AlphaT(const MCEventFormat*)`
`double AlphaT(const RecEventFormat*)`
The α_T variable [6].
`double EventMEFF(const MCEventFormat* event) const`
`double EventMEFF(const RecEventFormat* event) const`
The effective mass of the event.

`double EventMET(const MCEventFormat* event) const` `double EventMET(const RecEventFormat* event) const`	The event missing transverse energy.
`double EventMHT(const MCEventFormat* event) const` `double EventMHT(const RecEventFormat* event) const`	The event missing transverse hadronic energy.
`double EventTET(const MCEventFormat* event) const` `double EventTET(const RecEventFormat* event) const`	The event total transverse energy.
`double EventTHT(const MCEventFormat* event) const` `double EventTHT(const RecEventFormat* event) const`	The event total transverse hadronic energy.
`double MT2(const MALorentzVector* p1, const MALorentzVector* p2, const MALorentzVector& met, const double &mass)`	The event m_{T2} variable computed from a system of two visible objects `p1` and `p2`, the event missing momentum `met` and a test mass `mass` [4, 5].
`double MT2W(std::vector<const RecJetFormat*> jets, const RecLeptonFormat* lep, const ParticleBaseFormat& met)` `double MT2W(std::vector<const MCParticleFormat*> jets const MCParticleFormat* lep,const ParticleBaseFormat& met)`	The event m_{T2}^W variable computed from a system of jets `jets`, a lepton `lep` and the missing momentum `met` [2].

7 Signal regions, histograms and cuts

The implementation of an analysis in MadAnalysis 5 requires to deal with signal regions, selection cuts and histograms. Each analysis comes with an instance of the analysis manager class `RegionSelectionManager`, named `Manager()`, which allows the user to use the methods presented in the table below.

`void AddCut(const std::string&name, const std::string &RSname)`
`template<int NRS> void AddCut(const std::string&name, std::string const(& RSnames)[NRS])`
`void AddCut(const std::string &name)`
	Declares a cut named `name` and associates it with one region (the second argument is a string), with a set of regions (the second argument is an array of strings) or with all regions (the second argument is omitted).

`void AddHisto(const std::string&name,unsigned int nb,`
`  double xmin, double xmax)`
`void AddHistoLogX(const std::string&name,unsigned int nb,`
`  double xmin, double xmax)`

Declares a histogram named `name` of `nb` bins ranging from `xmin` to `xmax`. The histogram is associated with all regions and the x-axis can rely on a logarithmic scale (the second method).

`void AddHisto(const std::string&name,unsigned int nb,`
`  double xmin, double xmax, const std::string &RSname)`
`void AddHistoLogX(const std::string&name,unsigned int nb,`
`  double xmin, double xmax, const std::string &RSname)`

Same as above but the histogram is associated with a single region `RSname`.

`template <int NRS> void AddHisto(const std::string&name,`
`  unsigned int nb, double xmin, double xmax,`
`  std::string const(& RSnames)[NRS])`
`template <int NRS> void AddHistoLogX(const std::string&name,`
`  unsigned int nb, double xmin, double xmax,`
`  std::string const(& RSnames)[NRS])`

Same as above but the histogram is associated with an array of regions.

`void AddRegionSelection(const std::string& name)`

Declares a new region named `name`.

`bool ApplyCut(bool cond, std::string const &name)`

Applies the cut `name`, an event passing this cut if the condition `cond` is realized. The method returns `true` if at least one region is passing all cuts applied so far, or `false` otherwise.

`void FillHisto(std::string const&name, double val)`

Fills the histogram named `name`, the bin choice being driven by the value `value`.

`void InitializeForNewEvent(double EventWeight)`

To be called at the beginning of the analysis of an event in order to tag all regions as surviving the cuts and initialize the event weight to the value `EventWeight`.

`bool IsSurviving(const std::string &RSname)`

Verifies whether the region `RSname` survives all cut applied so far.

`void SetCurrentEventWeight(double weight)`

Modifies the weight of the current event to the value `weight`.

8 Message services

SampleAnalyzer handles four levels of streamers, that can be cast within any analysis code by typing one of the following lines,

```
INFO    << "..." << endmsg;
WARNING << "..." << endmsg;
ERROR   << "..." << endmsg;
DEBUG   << "..." << endmsg;
```

This allows the user to print informative, warning, error and debugging messages. Additionally, warning and error messages return information on the line number responsible for printing the message. The effect of a given message service can be modified by means of the methods presented in the table below.

`void DisableColor()`	Switches off the colored display of messages (that is on by default).
`void EnableColor()`	Switches on the colored display of messages.
`void SetMute()`	Switches entirely off a given message service (that is on by default).
`void SetStream(std::ostream* stream)`	Redirects the output of a given service to a file.
`void SetUnMute()`	Switches on a specific message service.

9 Sorting particles and objects

It is usually important to order particle as a function of one of their properties, like their transverse momentum or their energy. For this reason, SampleAnalyzer contains a series of routines allowing to sort a set of objects, which can be called by implementing

```
SORTER->sort(objects, criterion)
```

where `objects` is the vector of objects that needs to be sorted and `criterion` is the ordering variable. The latter can be `ETAordering` (pseudorapidity), `ETordering` (transverse energy), `Eordering` (energy), `Pordering` (the magnitude of the three-momentum), `PTordering` (the transverse momentum), `PXordering` (the x-component of the momentum), `PYordering` (the y-component of the momentum) and `PZordering` (the z-component of the momentum). As a result, the vector of objects is sorted by decreasing values of the ordering variable.

References

[1] J. R. Andersen et al. Les Houches 2013: Physics at TeV Colliders: Standard Model Working Group Report. 2014.

[2] Yang Bai, Hsin-Chia Cheng, Jason Gallicchio, and Jiayin Gu. Stop the Top Background of the Stop Search. *JHEP*, 07:110, 2012.

[3] R. Brun and F. Rademakers. ROOT: An object oriented data analysis framework. *Nucl. Instrum. Meth.*, A389:81–86, 1997.

[4] Hsin-Chia Cheng and Zhenyu Han. Minimal Kinematic Constraints and m(T2). *JHEP*, 12:063, 2008.

[5] C. G. Lester and D. J. Summers. Measuring masses of semiinvisibly decaying particles pair produced at hadron colliders. *Phys. Lett.*, B463:99–103, 1999.

[6] Lisa Randall and David Tucker-Smith. Dijet Searches for Supersymmetry at the LHC. *Phys. Rev. Lett.*, 101:221803, 2008.